Docket No. 042390.P9574
Express Mail No. EM560344452US

UNITED STATES PATENT APPLICATION

FOR

USING "SILENT STORE" INFORMATION TO ADVANCE LOADS

INVENTORS:

Adi Yoaz
Ronny Ronen
Rajesh Patel

PREPARED BY:

BLAKELY SOKOLOFF TAYLOR & ZAFMAN
12400 Wilshire Blvd., 7th Floor
Los Angeles, CA 90025-1026
(310) 207-3800

BACKGROUND OF THE INVENTION

FIELD OF THE INVENTION

This invention relates to improving processor performance, and more particularly to a method and apparatus for predicting silent stores and bypassing prior issued store instructions with unexecuted load instructions.

BACKGROUND INFORMATION

Out-of-order processors need to obey data dependancies. Register dependancies are easily identified by register names. Memory dependancies, however, are difficult to identify since the addresses of load and store instructions are only known at the time of execution. A load instruction typically transfers data from memory to a general register, a floating-point register, or a pair of floating-point registers. Store instructions typically transfer data from a general or floating-point register to memory. Processors, such as the P6 family, assume that load instructions depend on all previous store instructions issued, but not yet executed. Therefore, no load instructions can be advanced ahead of a previous issued store address operation. This restricts the scheduler by introducing false dependencies which, results in loss of performance.

Memory disambiguation is the ability to resolve whether a load address and a store address refer to the same address, i.e. the addresses would collide, or whether the addresses are different. This prediction would allow the advancing of load instructions before a store instruction if it is predicted not to collide.

Some of the previous approaches include the following. A scheme to never advance load instructions before a previous issued store instruction.

Also, there are various types of hardware memory, software assisted memory, and architecture assisted memory disambiguation schemes. Also, several approaches to memory disambiguation have been introduced to try to predict when an address of a load instruction and an address of a previously issued store instruction will differ.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 illustrates a system having an embodiment of the invention within a processor.

Figure 2 illustrates an embodiment of the invention having a silent store predictor.

Figure 3 illustrates a full collision history table (CHT).

Figure 4 illustrates an implicit-predictor CHT.

Figure 5 illustrates a tagless CHT.

Figure 6 illustrates a full CHT having distance bits.

Figure 7 illustrates an implicit-predictor CHT having distance bits.

Figure 8 illustrates a tagless CHT having distance bits.

Figure 9 illustrates a path update using an XOR.

Figure 10 illustrates a predictor indexing using an XOR.

Figure 11 illustrates an extended load buffer.

Figure 12 illustrates a flow diagram of a load instruction sequence of an embodiment of the invention.

Figure 13 illustrates a flow diagram of a store instruction sequence of an embodiment of the invention.

DETAILED DESCRIPTION

The invention generally relates to a method and apparatus for improving processor performance by predicting silent stores and bypassing the silent stores with unexecuted issued load instructions. Referring to the figures, exemplary embodiments of the invention will now be described. The exemplary embodiments are provided to illustrate the invention and should not be construed as limiting the scope of the invention.

A silent store is a store instruction that does not change the value already present at the target address. In other words, the value being written by the store instruction matches the exact value already stored at that same memory location. **Figure 1** illustrates system 100 having processor 110 including cache 105 and processing section 150 including predictor 140, a marking function, a comparing function, and a recovery mechanism. Also included in system 100 is memory 120, memory controller 130 and a bus. **Figure 2** illustrates an embodiment of the invention having processing section 150 including instruction fetcher and decoder 205, reorder buffer 210, retirement unit 220, memory and registers 230, controller 240, collision history table (CHT) 250 and execution unit 260.

According to one embodiment, predictor 140 uses CHT 250 to bypass a silent store. In this embodiment, CHT 250 is a memory storage device having a lookup table for making predictions about load/store instructions from an instruction sequence. A load that is bypassed is marked as such. The load is checked for disambiguation and it is determined whether the stored data is equivalent to the data in a load instruction, i.e. a silent store. If the data is equivalent, then the load can bypass the silent store.

In another embodiment, CHT 250 is used to predict whether a store instruction is silent. The store instruction is predicted as being silent when the store instruction, at some earlier time, was found to be silent. One should note that using CHT 250 to predict whether a store instruction is silent is only one way of predicting silent store instructions.

CHT 250 may have a variety of organizations and may use a variety of lookup and refreshment methods that are well known in the art of cache memories. **Figures 3-7** illustrate examples of CHTs. For example, CHT 250 can have entries that are indexed by a tag. The tag is searched when looking up a store instruction in CHT 250. **Figure 3** illustrates one embodiment where the tag of CHT 250 is linear instruction pointer (LIP) 310 for the store instruction. **Figure 4** illustrates another embodiment with CHT 250 where the entries of full and implicit predictor CHT 400 may be organized like entries of an n-way associative cache memory. **Figure 5** illustrates an embodiment where CHT 250 is tagless and is like a direct-mapped cache memory. In tagless CHT 250, each entry is indexed by a subset of the bits of the linear instruction pointer of the entry. The position of the entry in tagless CHT 250 also ordinarily contains information such as the instruction's position on a page of the memory, i.e. information from the remaining bits of the linear instruction pointer.

Different CHT's are capable of representing predictions for store instructions in different ways. In implicit-predictor type of CHT 250 illustrated in **Figure 4**, all entries in CHT 250 implicitly correspond to store instructions predicted to be colliding. In full and tagless CHT's, illustrated in **Figures 3** and **5**, resepctively, each entry has a respective predictor. The predictor bit is a bit that may be in a first or a second state. If the predictor bit is in the first state, the corresponding store instruction is predicted to be silent. If the predictor bit is in the second state, the corresponding store instruction is predicted to not be silent.

In other CHTs, entries may have additional bits corresponding to distances. **Figure 6** illustrates full CHT 250 having distance bits 610. **Figure 7** illustrates implicit-predictor CHT 250 having distance bits 710. **Figure 8** illustrates tagless CHT 250 having distance bits 810. The distance bits predict the distance, i.e. the number of instructions, that the execution of the store instruction associated with the entry can be advanced without the execution occurring before an earlier store instruction having the same data address. The distance may also be counted as the number of all instructions, such as the number of all store instructions. Generally, embodiments of CHT 250 only make predictions about store instructions that are simultaneously in a reorder buffer (see below).

The predictor and distance bits may take any of a variety of forms, including the predictor bits being sticky bits or saturating counters. The sticky bit cannot be reset to the second state after it has been set to the first state. With a sticky bit, once an instruction is predicted, the prediction is not updated. Since each entry of a tagless CHT corresponds to several store instructions, collision probabilities of entries may change substantially in time, and the sticky bit that cannot be updated is ordinarily not the appropriate predictor for a tagless CHT. The saturating counter is similar to the sticky bit except that a subsequent update may return the counter to the second state. The distance bits predict how far a store instruction can be advanced in the execution sequence and may take a variety of forms. In some embodiments, there may be enough distance bits to distinguish all distances between instructions in reorder buffer 210. In one embodiment, the distance bits give a refined prediction for when store instructions are not silent. In another embodiment, store instructions not executed more than the distance represented by the distance bits in advance of the location of the store instruction in the instruction sequence are predicted to be not silent. In another embodiment, the distance bits are updated to predict a shortest distance that the execution of the store instruction can be advanced without leading to a collision. In another embodiment, if the last store instruction in the distance is a silent store instruction, then the silent store instruction and other prior silent store instructions can also be bypassed.

The above discussed binary predictors, i.e., branch predictors. It should be noted that with branch predictors, the number of stores that are found to be predicted as silent which, are actually found to be not silent need to be minimal due to a cost of time. The results of the branch type of predictors may not be optimum. That is, too many non-silent stores are predicted as silent (1 bit, 2 bits) or only very few silent stores are predicted as such.

In another embodiment, CHT 250 is used as a silent store predictor and uses path based indexing. One should note that using CHT 250 as a silent store predictor using path based indexing is only one way of predicting silent store instructions. **Figure 9** illustrates a path update mechanism. Path based indexing is accomplished by performing an XOR on the store IP with encoded control flow information. The path is based on branches, not store instructions. In one embodiment, the path is encoded as follows. The path consists of a predefined number of bits n, such as n=16 bits. On any new taken branch, the accumulated path is left-shifted by s bits, such as s=2 bits. When a store is to be predicted, an XOR is performed on the store Ip with the path to index the appropriate state machine. This path encoding allows for aging of the path as well as coverage of the state machine tables. One skilled in the art will note that other indexing techniques can also be implemented. Predictor indexing is illustrated in **Figure 10**.

In one embodiment, the state machine can be such as a 1-bit, 2-bit, or sticky bit implementation. The type of state machine depends on the cost of misprediction. One-bit state machines predict more silent stores than other types of state machines, but more mistakes of predicting non-silent stores as silent are made. Two-bit and sticky bit state machines predict less silent stores than one-bit state machines and also make less mistakes, i.e. less mispredictions of silent stores. Other embodiments may include state machines that may use other parameters such as initial states of weak not-silent and strong not-silent. In a weak not-silent state, a parameter of silent or non-silent may be used. Tagged or tagless state machines may also be used in other embodiments.

The path updates are performed speculatively according to the speculated fetching of instruction. It should be noted that each branch has to be associated with its appropriate path. The path is recovered upon branch misprediction recovery. The state machines are then updated when the store commits.

After the silent store instructions are predicted, future load instructions can bypass the issued silent store instructions. For recovery purposes, the load instructions that are to bypass the silent store instructions are marked as bypassing. In one embodiment, the marking of the bypassing load is accomplished by the setting of one bit of the load predict portion in extended load buffer 1100 that is illustrated in **Figure 11**. In another embodiment, the bypassing load marking bit can also be not set in load predict segment 1120 to indicate a bypassing load. The predicted silent store needs to be marked as bypass. The marking of a silent store is accomplished by setting a bit in a store buffer (not shown). In another embodiment, the silent store marking bit can also be not set to indicate a silent store.

In another embodiment, since the store instructions will always snoop extended load buffer 1100 to ensure there are not younger load instructions that are completed and that match the store instructions memory address and data content, load instructions do not need to be marked since the marking of the load predict segment 1120 is redundant.

Upon an actual store instruction being executed, the value of the bypassing load instruction is then compared with the value of the data actually stored via the executed store instruction. If there is a mismatch, i.e., the value of the stored data and the bypassing load data are not the same, recovery begins. Thus, verification is performed by the store instruction snoops extended load buffer 1100 and the address and data are compared for the load instructions. This verifies that the load instruction has correct data.

With the recovery mechanism, if a predicted silent store instruction is found to be non-silent, then all bypassing load instructions and their associated dependent instructions have to be re-executed. In another embodiment, recovery of other instructions, such as only recovery including re-execution of advanced loads and their associated dependent instructions.

Figure 12 illustrates the flow for load instructions in one embodiment. An instruction is fetched by process 1210. It is then determiend whether the fetched instruction is a load instruction or not by process 1220. If the instruction fetched is not a load instruction, process 1200 exits at 1280. If the fetched instruction is a load instruction, the load is scheduled for execution by process 1230. Silent store instructions are then bypassed and the load is executed by process 1240. It is then determined if the load instruction is completed by process 1250. If the load instruction is not complete, the load instruction is continued to be executed by process 1240. If the load instruction is complete, the load is prepared for retirment by process 1260. It is then determined if the load is marked flush by process 1270. If the load is marked flush, then process 1200 continues at start 1205 where process 1210 fetches the next instruction. If the load is not marked flush, process 1200 the proceeds to exit 1280.

Figure 13 illustrates the flow for store instructions in one embodiment. An instruction is fetched by process 1310. It is determined if the instruction is a store by process 1320. If the fetched instruction is not a store, process 1300 exits at 1399. If the fetched instruction is a store instruction, a predictor is processed for predicting silent stores by process

1330. The store instruction is then scheduled for execution by process 1340. The store instruction is then executed by process 1350. The load buffer is snooped by process 1360 next. The address and data of the store instruction are then compared with all executed load instructions in extended load buffer 1100 by process 1370. Process 1380 determines if a match occurs. If a match occurs, process 1390 sets load address match segment, load data matched segment, and load flush segment in extended buffer 1100. If a match does not occur, process 1300 proceeds with store retirement process 1396. It is determined then if the store instruction can be retired by process 1397. If the store instruction can be retired, process 1398 then updates a cache with the store instruction and then proceeds to exit 1399. If the store instruction can not be retired, process 1300 continues to start 1305.

For ease of explanation, an example will be presented describing how extended load buffer 1100 enables misprediction detection. Assuming the following instructions are issued in the order presented:

St0 A [data D1] (Store instruction to store data D1 from register 0 to address A)

St1 A [data D1] (Store instruction to store data D1 from register 1 to address A)

Ld0 A (Load instruction to load the data D1 from address A, to register 0)

If instructions St0 and St1 are not executed yet, and predicted as silent, then Ld0 will bypass St0 and St1 ignoring the fact that their addresses are unknown. Ld0 receives data from the cache/memory structure and writes the data onto the write-back bus and into the load buffer. Upon St0 being executed, its address and data are compared against all previous younger loads from St0 load instructions in the load buffer. If the address and data are valid in the load buffer, and the store address and data match the load, then the load has the correct value. If the address matches, but the loaded data does not match the store data, the load and its dependent instructions must be re-executed, or be flushed out in case there is no re-execution mechanism in the embodiment. Thus, in the above example, the address and data for St0 and St1 match the value loaded by Ld0. The bypass is correct and performance is gained due to earlier execution of the load.

The content of the load buffer after the execution of Ld0, St0 and St1 would be as follows:

Load address buffer:	A
Load Attribute buffer:	n/a
Load Data Buffer:	D1
Address content addressable memory (CAM) match:	1
Data CAM match:	1
Load Flush:	0

Also for ease of discussion, another example is presented below for which a misprediction is detected. For this example, the following instructions are listed in the issued order:

St0 A [data D1] (Store instruction to store data D1 from register 0 to address A)

St1 A [data D2] (Store instruction to store data D2 from register 1 to address A)

Ld0 A (Load instruction to load the data from address A, D2, to register 0)

For this example, assume that St0 and St1 have not executed yet. Ld0 gets data D1 from the cache/memory structure and writes it onto the write-back bus and into the load buffer. Upon St0 being executed, its address and data are compared against all previous ("younger") loads in the load buffer. If the load data is valid, the address and data will match with the load and no action is taken. Upon St1 being executed, its address and data are compared. In St1's case, however, the data did not match and the load flush bit is set. Once the load is "tagged" as a load flush, it acts as a sticky bit and can not be reset. If the load data is not valid (does not match) and the address matches, then load flush is set. For OOO (out of order) machines, if St1 executes before St0, load flush will be set. Upon St0 executed, the load flush is not reset since the load flush is sticky.

The content of the load buffer after the execution of Ld0, St0 and St1 would be as follows:

Load address buffer:	A
Load Attribute buffer:	n/a
Load Data Buffer:	D1
Address content addressable memory (CAM) match:	1
Data CAM match:	0
Load Flush:	1

For an implementation where the store instruction is broken into two microinstructions (store address and store data), such as in P1U and P6, the address and data comparisons may occur at different times, but the mechanism to set load flush remains the same.

Therefore, embodiments of the invention expands cases where a load instruction can bypass previous issued store instructions. The process of memory disambiguation is enhanced by allowing loads to bypass silent stores without being flushed. The number of flushes caused by a load bypassing an older store with the same address is reduced. Thus, improving processor performance. Also, one should note that even without predicting silent store instructions, that other features are present. Since silent stores can be marked as such post facto, embodiments can take advantage of this knowledge. One such advantage is bypassing with other load instructions. Another advantage is ignoring the writing since the data is equivalent. Thus saving amount of writes and improving performance.

The above embodiments can also be stored on a device or medium and read by a machine to perform instructions. The device or medium may include a solid state memory device and/or a rotating magnetic or optical disk. The device or medium may be distributed when partitions of instructions have been separated into different machines, such as across an interconnection of computers.

While certain exemplary embodiments have been described and shown in the accompanying drawings, it is to be understood that such embodiments are merely illustrative of and not restrictive on the broad invention, and that this invention not be limited to the specific constructions and arrangements shown and described, since various other modifications may occur to those ordinarily skilled in the art.